AtomC - generarea de cod

```
unit:
    {
    addInstr(OP_CALL);
    addInstr(OP_HALT);
    }
    ( structDef | fnDef | varDef )* END
    {
    Symbol *sm=findSymbol("main");
    if(!sm)tkerr(iTk,"undefined: main");
    instructions[0].arg.i=sm->fn.instrIdx;
    }
```

```
fnDef: ( typeBase | VOID ) ID
    LPAR ( fnParam ( COMMA fnParam )* )? RPAR
    {
    owner->fn.instrIdx=nInstructions;
    addInstr(OP_ENTER);
    }
    stmCompound[false]
    {
    instructions[owner->fn.instrIdx].arg.i=symbolsLen(owner->fn.locals);
    if(owner->type.tb==TB_VOID)addInstrWithInt(OP_RET_VOID,symbolsLen(owner->fn.params));
    /* owner=NULL; */
    }
```

```
stm: stmCompound
           | IF LPAR expr RPAR
                {
                addRVal(rCond.lval,&rCond.type);
                Type intType={TB_INT,NULL,-1};
                insertConvIfNeeded(nInstructions,&rCond.type,&intType);
                int posJF=addInstr(OP_JF);
                }
                stm ( ELSE
                    {
                    int posJMP=addInstr(OP_JMP);
                    instructions[posJF].arg.i=nInstructions;
                    }
                    stm {instructions[posJMP].arg.i=nInstructions;} |
                    {
                    instructions[posJF].arg.i = nInstructions;
                    } )
           | WHILE {int posCond=nInstructions;} LPAR expr RPAR
                {
                addRVal(rCond.lval,&rCond.type);
                Type intType={TB_INT,NULL,-1};
                insertConvIfNeeded(nInstructions,&rCond.type,&intType);
                int posJF=addInstr(OP_JF);
                }
```

```
            stm
            {
            addInstrWithInt(OP_JMP,posCond);
            instructions[posJF].arg.i=nInstructions;
            }
        | FOR LPAR expr? SEMICOLON expr? SEMICOLON expr? RPAR stm
        | BREAK SEMICOLON
        | RETURN ( expr
            {
            addRVal(rExpr.lval,&rExpr.type);
            insertConvIfNeeded(nInstructions,&rExpr.type,&owner->type);
            addInstrWithInt(OP_RET,symbolsLen(owner->fn.params));
            } | {addInstr(OP_RET_VOID);} ) SEMICOLON
        | (expr {if(rExpr.type.tb!=TB_VOID)addInstr(OP_DROP);} )? SEMICOLON
```

```
exprAssign: exprUnary ASSIGN exprAssign
    {
    addRVal(r->lval,&r->type);
    insertConvIfNeeded(nInstructions,&r->type,&rDst.type);
    switch(rDst.type.tb){
        case TB_INT:addInstr(OP_STORE_I);break;
        case TB_DOUBLE:addInstr(OP_STORE_F);break;
        }
    }
    | exprOr
```

```
exprRel: {Token *op;} exprRel ( LESS[op] | LESSEQ[op] | GREATER[op] | GREATEREQ[op] )
    {
    int posLeft=nInstructions;
    addRVal(r->lval,&r->type);
    }
    exprAdd
    {
    addRVal(right.lval,&right.type);
    insertConvIfNeeded(posLeft,&r->type,&tDst);
    insertConvIfNeeded(nInstructions,&right.type,&tDst);
    switch(op->code){
        case LESS:
            switch(tDst.tb){
                case TB_INT:addInstr(OP_LESS_I);break;
                case TB_DOUBLE:addInstr(OP_LESS_F);break;
                }
            break;
        }
    /* *r=(Ret){{TB_INT,NULL,-1},false,true}; */
    }
    | exprAdd
```

```
exprAdd: {Token *op;} exprAdd ( ADD[op] | SUB[op] )
    {
    int posLeft=nInstructions;
    addRVal(r->lval,&r->type);
```

```
    }
    exprMul
    {
    addRVal(right.lval,&right.type);
    insertConvIfNeeded(posLeft,&r->type,&tDst);
    insertConvIfNeeded(nInstructions,&right.type,&tDst);
    switch(op->code){
        case ADD:
            switch(tDst.tb){
                case TB_INT:addInstr(OP_ADD_I);break;
                case TB_DOUBLE:addInstr(OP_ADD_F);break;
            }
            break;
        case SUB:
            switch(tDst.tb){
                case TB_INT:addInstr(OP_SUB_I);break;
                case TB_DOUBLE:addInstr(OP_SUB_F);break;
                }
            break;
        }
    /* *r=(Ret){tDst,false,true}; */
    }
    | exprMul
```

```
exprMul: {Token *op;} exprMul ( MUL[op] | DIV[op] )
  {
  int posLeft=nInstructions;
  addRVal(r->lval,&r->type);
  }
  exprCast
  {
  addRVal(right.lval,&right.type);
  insertConvIfNeeded(posLeft,&r->type,&tDst);
  insertConvIfNeeded(nInstructions,&right.type,&tDst);
  switch(op->code){
    case MUL:
      switch(tDst.tb){
        case TB_INT:addInstr(OP_MUL_I);break;
        case TB_DOUBLE:addInstr(OP_MUL_F);break;
        }
      break;
    case DIV:
      switch(tDst.tb){
        case TB_INT:addInstr(OP_DIV_I);break;
        case TB_DOUBLE:addInstr(OP_DIV_F);break;
        }
    break;
  }
  /* *r=(Ret){tDst,false,true}; */
  }
  | exprCast
```

```
exprPrimary: ID[tkName] ( LPAR ( expr[&rArg]
  {
  addRVal(rArg.lval,&rArg.type);
  insertConvIfNeeded(nInstructions,&rArg.type,&param->type);
  /*param=param->next;*/
  }
    ( COMMA expr[&rArg]
      {
      addRVal(rArg.lval,&rArg.type);
      insertConvIfNeeded(nInstructions,&rArg.type,&param->type);
      /*param=param->next;*/
      }
      )* )? RPAR
  {
  if(s->fn.extFnPtr){
    int posCallExt=addInstr(OP_CALL_EXT);
    instructions[posCallExt].arg.extFnPtr=s->fn.extFnPtr;
    }else{
    addInstrWithInt(OP_CALL,s->fn.instrIdx);
    }
  }
  |
  {
  if(s->kind==SK_VAR){
    if(s->owner==NULL){    // variabile globale
      addInstrWithInt(OP_ADDR,s->varIdx);
      }else{    // variabile locale
      switch(s->type.tb){
        case TB_INT:addInstrWithInt(OP_FPADDR_I,s->varIdx+1);break;
        case TB_DOUBLE:addInstrWithInt(OP_FPADDR_F,s->varIdx+1);break;
        }
      }
    }
  if(s->kind==SK_PARAM){
    switch(s->type.tb){
      case TB_INT:
addInstrWithInt(OP_FPADDR_I,s->paramIdx-symbolsLen(s->owner->fn.params)-1); break;
      case TB_DOUBLE:
addInstrWithInt(OP_FPADDR_F,s->paramIdx-symbolsLen(s->owner->fn.params)-1); break;
      }
    }
  }
  )
    | CT_INT[&ct]    {addInstrWithInt(OP_PUSH_I,ct->i);}
    | CT_REAL[&ct]    {addInstrWithDouble(OP_PUSH_F,ct->r);}
    | CT_CHAR[&ct]
    | CT_STRING[&ct]
    | LPAR expr[r] RPAR
```